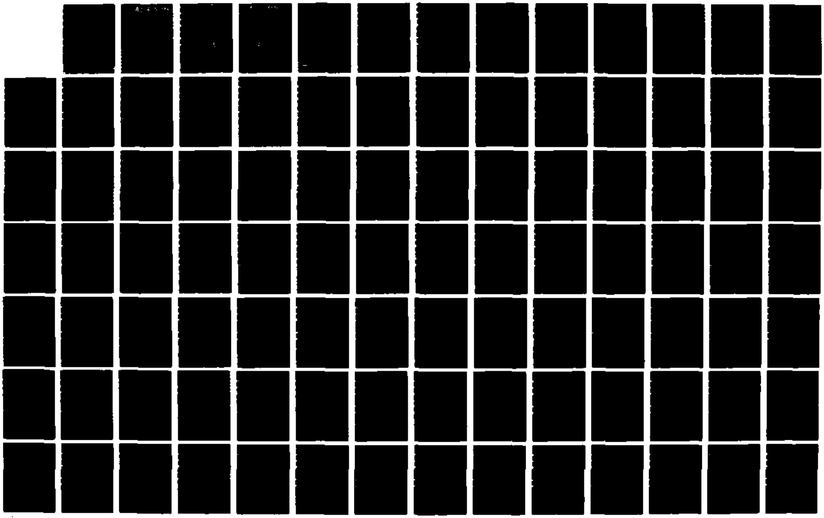
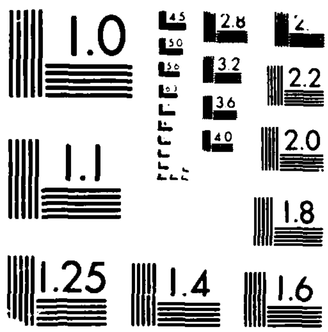
AD-A190 366 TUTORIAL TRACK II ADVANCED ADA TOPICS(U) INFORMATION SYSTEMS AND TECHNOLOGY CENTER W-P AFB OH ADA VALIDATION FACILITY P LAWLIS ET AL 09 JUN 87 1/2

UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

AD-A190 366

TUTORIAL

TRACK II

ADVANCED ADA TOPICS

By

Major Patricia Lawlis, Air Force Institute of Technology

and

Captain Dean Gonzalez, U.S. Air Force Academy

and

Lieutenant David Cook, U.S. Air Force Academy

UNCLASSIFIED

DTIC FILE COPY

SECURITY CLASSIFICATION

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETEING FORM

AD-A190 366

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Tutorial Track II. Advanced Ada Topics

5. TYPE OF REPORT & PERIOD COVERED

Tutorial, 9 June, 1987

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

MAJ Patricia Lawlis, CAPT Dean Gonzalez, and LT David Cook

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

Ada Software Education and Training Team
Ada Joint Program Office, 3E114, The
Pentagon, Washington, D.C.20301-3081

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
3E 114, The Pentagon
Washington, DC 20301-3081

12. REPORT DATE

June 9, 1987

13. NUMBER OF PAGES

44

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)

Ada Joint Program Office

15. SECURITY CLASS (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION DOWNGRADING SCHEDULE

N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC ELECTE JAN 0 6 1988 S D

D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language; Ada Training; Education, Training, Computer Programs, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document contains prints of viewgraphs presented at the Advanced Ada Topics Tutorial, Track II June 9, 1987. Topics covered were: Data Abstraction, Tasking, Strong Typing, and Exceptions

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Tasking*

Abstraction of Process

by

Dean W. Gonzalez

David A. Cook

303-472-2136
AV 259-2136

U.S. Air Force Academy

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

ADA TASKING

- OVERVIEW

  DEFINE ADA TASKING

  DEFINE SYNCHRONIZATION MECHANISM

  EXAMPLES

- STARTED AFTER ELABORATION OF PARENT, AND BEFORE THE PARENT'S FIRST STATEMENT

- MAY ALSO BE A TYPE AND TREATED AS AN OBJECT

Ada Tasking

Task Definition

- A program unit for concurrent execution
- Never a Library Unit
- Master is a ...
  - Library Package
  - Subprogram
  - Block Statement
  - Other Task

Callee Provides Service

1. Immediate Response
2. Wait for a while
3. Wait forever

Service is Requested with an entry call statement

Service is provided with an accept statement

ADA TASKING

SYNCHRONIZATION MECHANISMS

- GLOBAL VARIABLES
- RENDEZVOUS

  MAIN PROGRAM IN A TASK

  CALLER REQUESTS SERVICE

  1. IMMEDIATE REQUEST
  2. WAIT FOR A WHILE
  3. WAIT FOREVER

ADA TASKING

SCENARIO I

"THE GOLDEN ARCHES"

MCD TASKS :

- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED : NONE

GONZO TASKS :

- SERVICE PROVIDED : NONE
- SERVICE REQUESTED : FOOD

ADA TASKING

- SELECT STATEMENT PROVIDES ABILITY TO PROGRAM THE DIFFERENT 'REQUEST' AND 'PROVIDE' MODES
- GUARDS ARE "IF STATEMENTS" FOR THE PROVIDING SERVICE
- TERMINATION IS AN ALTERNATIVE IF A SERVICE IS NO LONGER NEEDED

```
Task McD is
   entry SERVE(TRAY_OF : out FOOD_TYPE);
end McD;

Task GONZO;

Task Body McD is
   NEW_TRAY : FOOD_TYPE;
   function COOK return FOOD_TYPE is.......
   begin
      loop
         accept SERVE (TRAY__OF : out FOOD_TYPE) do
            TRAY_OF := COOK;
         end;
      end loop;
   end McD;
```

```
Task Body GONZO is
   MY_TRAY : FOOD_TYPE;
   procedure CONSUME (MY_TRAY : in FOOD_TYPE) is ...
   begin
      loop
         McD.SERVE ( MY_TRAY );
         CONSUME (MY_TRAY);
      end loop;
   end GONZO;
```

```
Task Body McD is
   NEW_TRAY : FOOD_TYPE;


   function COOK return FOOD_TYPE is

      ...
   end COOK;


   begin
      loop
         NEW_TRAY := COOK;
         accept SERVE (TRAY_OF : out FOOD_TYPE ) do
            TRAY_OF := NEW_TRAY;
         end SERVE;

      end loop;
  end GONZO;
```

```
loop

   NEW_TRAY := COOK;
   select
      accept SERVE (TRAY_OF : out FOOD_TYPE) do...
         TRAY_OF := NEW_TRAY;
      end SERVE;
   else
      null;
   end select;

end loop;
```

```
loop

   NEW_TRAY := COOK;
   select
      accept SERVE (TRAY_OF : out FOOD_TYPE) do...
         TRAY_OF := NEW_TRAY;
      end SERVE;
   else
      terminate;
   end select;

end loop;
```

```
loop

   NEW_TRAY := COOK;
   select
      accept SERVE (TRAY_OF : out FOOD_TYPE) do...
         TRAY_OF := NEW_TRAY;
      end SERVE;
   or
      delay 15 * MINUTES;
   end select;

end loop;
```

```
loop

   select
      McD.SERVE(MY_ORDER); consume (MY_ORDER);
   else
      select
         BK.SERVE(MY_ORDER); consume (MY_ORDER);
      else
         exit;
      end select;
   end select;

end loop;
```

```
loop
   select
      McD.SERVE(MY_ORDER);  consume (MY_ORDER);
   or
      delay 10.0 * MINUTES;
      select
         BK.SERVE(MY_ORDER);  consume (MY_ORDER);
      or
         delay 5.0 * MINUTES;
         exit;
      end select;
   end select;
end loop;
```

```
loop

   select
      McD.SERVE (MY_ORDER);
   or
      BK.SERVE (MY_ORDER);
   end select;

   consume;

end loop;
```

```
loop

   select
      McD.SERVE (MY_ORDER);
   or
      BK.SERVE (MY_ORDER);

   else
      delay 10 * MINUTES;
      exit;
   end select;

   consume;


end loop;
```

ADA TASKING

SCENARIO II

"NO FREE LUNCH"

McD TASK

SERVICE PROVIDED : FOOD
SERVICE REQUESTED: MONEY

GONZO TASK

SERVICE PROVIDED : MONEY
SERVICE REQUESTED: FOOD

```
Task McD is
   entry SERVE ( ORDER : out FOOD_TYPE;
                 COST  : in  MONEY_TYPE);
end McD;

TASK GONZO;
```

--OR

```
Task McD is
   entry SERVE ( ORDER : out FOOD TYPE):
end McD;


Task GONZO is
   entry PAY ( COST    :  in MONEY_TYPE;
               PAYMENT : out MONEY_TYPE);
end GONZO;
```

```
Task Body McD is
     CASH_DRAWER : MONEY_TYPE;
     NEW_ORDER   : FOOD_TYPE;
     function COOK .................
     function CALC_COST (ORDER : in FOOD_TYPE )
              return MONEY_TYPE is ........
   begin
     loop
        NEW_ORDER := COOK;
        select
           accept SERVE(ORDER : out FOOD_TYPE) do
              ORDER := NEW_ORDER:
              COST := CALC_COST (NEW_ORDER);
              GONZO_PAY (COST, AMOUNT_PAID);
              CASH_DRAWER := CASH_DRAWER + AMOUNT_PAID;
              end SERVE;
         or
              delay 15.0 * MINUTES;
        end select;
     end loop;
 end McD;
```

```
Task Body GONZO is
        ACCOUNT_BALANCE : MONEY_TYPE;
        MY_ORDER : FOOD_TYPE;
        function GO_TO_WORK return MONEY_TYPE is....
    begin
        ACCOUNT_BALANCE := GO_TO_WORK + ACCOUNT_BALANCE;
        loop
           McD.SERVE (MY_ORDER);
           accept PAY (COST    : in MONEY_TYPE;
                       PAYMENT : out MONEY_TYPE) do
              ACCOUNT_BALANCE := ACCOUNT_BALANCE -
                                   COST ;
              PAYMENT := COST;
           end PAY;
        end loop;
    end GONZO;
```

ADA TASKING

SCENARIO II A

"NO WAIT FOR THE WAITERS"

MCD TASK

SERVICE PROVIDED : FOOD
SERVICE REQUESTED: MONEY

GONZO TASK

SERVICE PROVIDED : MONEY
SERVICE REQUESTED: FOOD

MANAGER TASK

SERVICE PROVIDED : MAKE NEW WAITER
SERVICE REQUESTED: NONE

```
Task type McD is
    entry SERVE....
end McD;

Task GONZO is
    entry PAY....
end GONZO;

Task MANAGER;

Type CASHIER_POINTER is access McD;

Type REGISTER_TYPE is array (1..NO_REGISTERS)
                    of CASHIER_POINTER;

THE_REGISTERS : REGISTER_TYPE:=(others => new McD);
```

```
Task Body McD is
  ...
  ...
  ...
  begin
   loop
    NEW_ORDER := COOK;
    select
       accept SERVE.....
       ...
       end SERVE;
     or
       delay 2.0 * MINUTES;
       exit;
     end select;
   end loop;
```

```
Task Body GONZO is
   ...
   ...
   begin
     ...
     ...
     --Now, GONZO has to search for the open
     --  registers, and select the one with
     --  the shortest line
     ...
     ...
     THE_REGISTERS(MY_REGISTER).SERVE;
     ...
   end GONZO;
```

```
Task Body MANAGER is
  ...

  ...
  begin
    loop
   --The MANAGER will look at the queue lengths of
   --  the open registers, and, when necessary
   --  will open registers that are currently
   --  closed
   ...
      if ............ then
        THE_REGISTERS(CLOSED_REGISTER) := new McD;
      end if;
    end loop;
end MANAGER;
```

Ada Tasking

Scenario III

"A Sugar Cone, Please:

BR Task
- Service Provided : Ice Cream
- Service Requested: An Order

Servomatic Task
- Service Provided : A Number

Customers Task
- Service Provided : An Order
- Service Requested: Ice Cream

```
task BR is
   entry SERVE (ICE_CREAM : out DESSERT_TYPE);
end BR;

task SERVOMATIC is
   entry TAKE (A_NUMBER : out SERVOMATIC_NUMBERS);
end SERVOMATIC;

task type CUSTOMER_TASK is
   entry REQUEST (ORDER : out ORDER_TYPE);
end CUSTOMER_TASK;

type CUSTOMER is access CUSTOMER_TASK;

CUSTOMERS : array (SERVOMATIC_NUMBERS) of CUSTOMER;
```

```
task body BR
     NEXT_CUSTOMER : SERVOMATIC_NUMBERS :=
                SERVOMATIC_NUMBERS'last;
     CURRENT_ORDER : ORDER_TYPE;
     ICE_CREAM : DESSERT_TYPE;
     function MAKE (ORDER : in ORDER_TYPE) return
                             DESSERT_TYPE is......
  begin
     loop
        begin
           NEXT_CUSTOMER := (NEXT_CUSTOMER + 1)
                        mod SERVOMATIC_NUMBERS'last;
           CUSTOMERS(NEXT_CUSTOMER).REQUEST
                                 (CURRENT_ORDER);
           ICE_CREAM := MAKE(CURRENT_ORDER);
           accept SERVE(ICE_CREAM : out DESSERT_TYPE) do
                 ICE_CREAM := BR.ICE_CREAM;
           end SERVE;
        exception
           when TASKING_ERROR => null;
                 --customer not here
        end;
     end loop;
  end;
```

```
task body SERVOMATIC is
        NEXT_NUMBER : SERVOMATIC_NUMBERS :=
                               SERVOMATIC_NUMBERS'first;
    begin
        loop
            accept TAKE(A_NUMBER : out SERVOMATIC_NUMBERS) do
                A_NUMBER := NEXT_NUMBER;
            end TAKE;
            NEXT_NUMBER := (NEXT_NUMBER + 1) mod
                                          SERVOMATIC_NUMBERS'last;
        end loop;
    end SERVOMATIC;
```

```
task body CUSTOMER_TASK is
     MY_ORDER : ORDER_TYPE := ... --some value;
     MY_DESSERT : DESSERT_TYPE;
  begin
     accept REQUEST ( ORDER : out ORDER_TYPE) do
          ORDER := MY_ORDER;
     end REQUEST;
     BR.SERVE(MY_DESSERT);
     --eat the dessert, or do whatever
  end;
```

ADA TASKING

SCENARIO IV

"LETS HIDE THE SPOOLER TASK"

PRINTER_PACKAGE
ACTION-"HIDES" THE PRINT SPOOLER
BY RENAMING TASK ENTRY

SPOOLER TASK
SERVICE PROVIDED : VIRTUAL PRINT
SERVICE REQUESTED: PHYSICAL PRINT

PRINTER TASK
SERVICE PROVIDED : PHYSICAL PRINT
SERVICE REQUESTED: FILE NAME

```
Package PRINTER_PACKAGE is
   ...
   ...
   task SPOOLER is
      entry PRINT_FILE (NAME : in STRING;
                        PRIORITY : in NATURAL);
      entry PRINTER_READY;
   end SPOOLER;
   ...
   ...
   procedure PRINT ( NAME : in STRING;
                     PRIORITY : in NATURAL := 10)
      renames SPOOLER.PRINT_FILE;
end PRINTER_PACKAGE;


Package Body PRINTER_PACKAGE is
   ...
   ...
   task PRINTER is
      entry PRINT_FILE (NAME : in STRING );
   end PRINTER;
   ...
   ...
end PRINTER_PACKAGE;
```

```
task body SPOOLER is
    begin
      loop
       select
          accept PRINTER_READY do
             PRINTER.PRINT_FILE ( REMOVE (QUEUE)  );
             -- Remove would determine the next job and
             --    send it to the actual printer
          end PRINTER_READY;
       else
          null;
       end select;
       select
          accept PRINT_FILE ( NAME : in STRING;
                              PRIORITY : NATURAL ) do
             INSERT ( NAME, PRIORITY);
             --put name on queue or queues according
             --   to priority
          end PRINT_FILE;
        else
           null;
        end select;
     end select;
    end loop;
end SPOOLER;
```

```
task body PRINTER is
   begin
      loop
         SPOOLER.PRINTER_READY;
         accept PRINT_FILE ( NAME : in STRING ) do

            if NAME'length /= 0 then ........

               --print the file

            else
               delay 10.0 * seconds;
            end if;

          end PRINT_FILE;
       end loop;
   end PRINTER;
```

```
with PRINTER_PACKAGE;
procedure MAIN is
     .
     .
     .
     loop
          --process several files
          PRINTER_PACKAGE.PRINT (A_FILE, A_PRIORITY);
          .
          .
     end loop;
end MAIN;
```

APPLICATIONS FOR TASKS

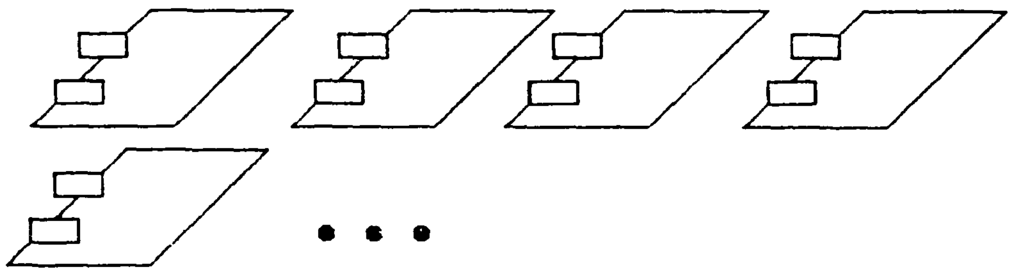
- CONCURRENT OPERATIONS
- ROUTING MESSAGES
- SHARED RESOURCE MANAGEMENT
- INTERRUPT HANDLING

MATRIX MULTIPLICATION

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 0 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

```
type ROW_OR_COL is array (integer range <>) of integer;
type PTR is access ROW_OR_COL;

task type PARTIAL is
   entry SEND  (ROW, COL : ROW_OR_COL);
   entry RECEIVE (RESULT : out integer);
end PARTIAL;
```

MAIN



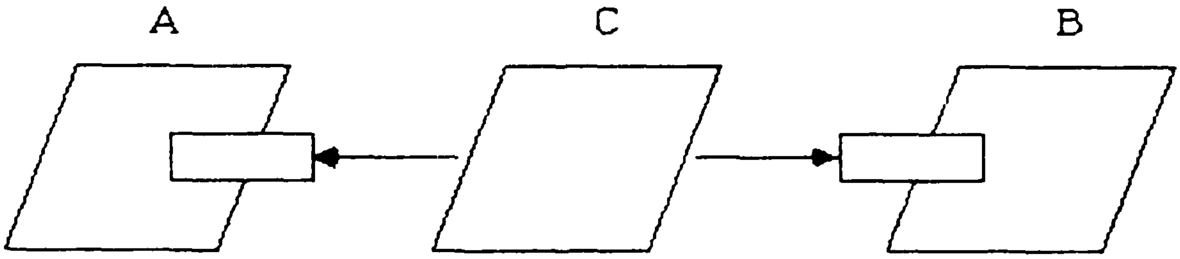
```
begin
  -- send row and col
  -- receive partial product
end
```

```
task body PARTIAL is

    PRODUCT : integer := 0;
    ROW_PTR : PTR;
    COL_PTR : PTR;

begin

    accept SEND (ROW,COL : ROW_OR_COL) do
        ROW_PTR := new ROW_OR_COL'(ROW);
        COL_PTR  := new ROW_OR_COL'(COL);
    end SEND;

    for I in ROW_PTR.all'range
    loop
       PRODUCT := PRODUCT +
                  ROW_PTR(I) * COL_PTR(I);
    end loop;

    accept RECEIVE (RESULT : out integer) do
        RESULT := PRODUCT;
    end RECEIVE;

end PARTIAL;
```

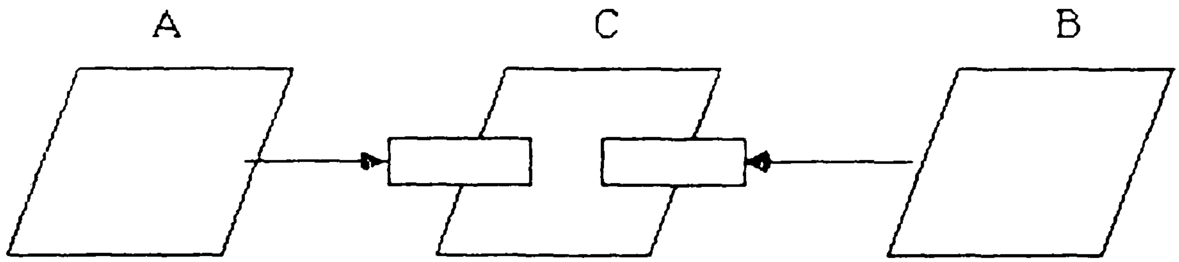
```
procedure MAIN is

 COLS : constant := 10;
 ROWS : constant := 10;
 type MATRIX is array (1 .. ROWS) of
                               ROW_OR_COL (1 .. COLS);

 MAT : MATRIX;
 VECTOR : ROW_OR_COL (1 .. COLS);
 FINAL    : ROW_OR_COL (1 .. ROWS);

 . . .
 declare

    WORKER : array (1 .. ROWS) of PARTIAL;  -- tasks

 begin
   for I in 1 .. ROWS
   loop
       WORKER(I).SEND(ROW => MAT(I),
                      COL => VECTOR);
   end loop;

   for I in 1 .. ROWS
   loop
       WORKER(I).RECEIVE (FINAL(I));
   end loop;

end;        -- block
. . .
```

ROUTING MESSAGES

- WRITE TASK SPECIFICATIONS TO SEND AN INTEGER FROM TASK A TO TASK B

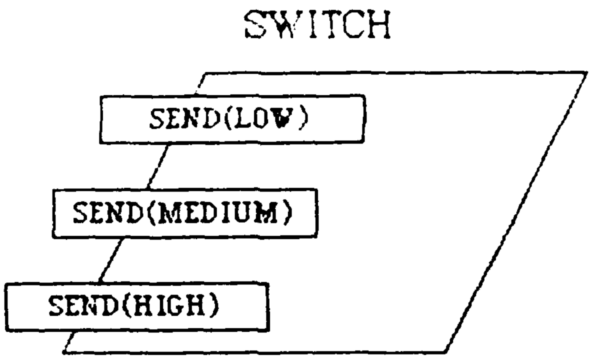


- WRITE SPECIFICATIONS AND BODIES FOR THE FOLLOWING SYSTEM. TASK C WILL REPEATEDLY GET AN INTEGER FROM TASK A AND SEND IT ON TO TASK B



PRIORITY MESSAGES

```
type PRIORITY is (LOW, MEDIUM, HIGH);

task SWITCH is
   entry SEND (PRIORITY)
          (M : in string);
end SWITCH;
```




```
task body SWITCH is
begin
  loop
    select

      accept SEND(HIGH) do . . . end SEND;

    or

      when SEND(HIGH)'count = 0 =>
      accept SEND(MEDIUM) do  . . . end SEND;

    or

      when SEND(HIGH)'count = 0 and
             SEND(MEDIUM)'count = 0 =>
      accept SEND (LOW) . . . end SEND;

    end select;
  end loop;
end SWITCH;
```

A Synchronizing Buffer

```
task SYNCHRONIZER is
  entry PUT (ITEM : in SOME_TYPE);
  entry GET (ITEM : out SOME_TYPE);
end SYNCHRONIZER;

task body SYNCHRONIZER is

   SPOT : SOME_TYPE;

begin

  loop

       accept PUT (ITEM : in SOME_TYPE) do
            SPOT := ITEM;
       end PUT;

       accept GET (ITEM : out SOME_TYPE) do
             ITEM := SPOT;
       end GET;

    end loop;

end SYNCHRONIZER;
```

CONTROLING RESOURCES

- SEVERAL CONCERNS ARE PRESENT WHEN DEALING WITH PARALLELISM THAT ARE NOT PRESENT WHEN DEALING IN A PURELY SEQUENTIAL MODE

- IT IS IMPORTANT TO BE ABLE TO ASSURE THAT A VALUE IS NOT BEING CHANGED BY ONE USER AT THE PRECISE MOMENT THAT IT IS BEING REFERENCED BY ANOTHER USER

- Ada PROVIDES A PRAGMA 'SHARED' WHICH CAN HELP

```
INDEX : integer;
pragma SHARED(INDEX);
```

- ENFORCES MUTUALLY EXCLUSIVE ACCESS

- AVAILABLE FOR SCALAR AND ACCESS TYPES ONLY

SEMAPHORES

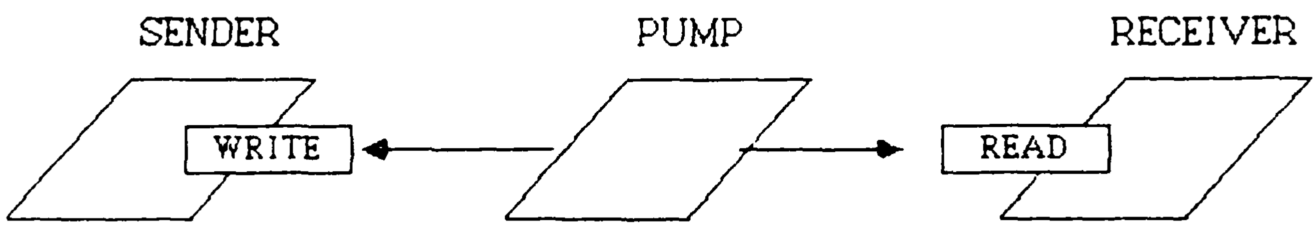
```
task SEMAPHORE is
     entry SEIZE;
     entry RELEASE;
end SEMAPHORE;

task body SEMAPHORE is
  IN_USE : boolean := false;
begin
  loop
     select

        when not IN_USE =>
        accept SEIZE do
             IN_USE := true;
        end SEIZE;

     or

        when IN_USE =>
        accept RELEASE do
             IN_USE := false;
        end RELEASE;

     end select;
  end loop;
end SEMAPHORE;
```

ENCAPSULATING A DATA ITEM

```
task PROTECTED is
     entry SET (OBJ : in integer);
     entry GET (OBJ : out integer);
end PROTECTED;

task body PROTECTED is
     LOCAL : integer;
begin
  loop
     select
          accept SET (OBJ : in integer) do
               LOCAL := OBJ;
          end SET;
     or
          accept GET (OBJ : out integer) do
               OBJ := LOCAL;
          end GET;
     end select;
  end loop;
end PROTECTED;
```

PUMPING TASK

```
task PUMP;

task SENDER is
   entry WRITE (ITEM : out SOME_TYPE);
end SENDER;

task RECEIVER is
   entry READ (ITEM : in SOME_TYPE);
end RECEIVER;

task body PUMP is
   THE_ITEM : SOME_TYPE;
begin
   loop
      SENDER.READ(THE_ITEM);
      RECEIVER.WRITE(THE_ITEM);
   end loop;
end PUMP;

task body SENDER is separate;
task body RECEIVER is separate;
```



HARDWARE INTERRUPTS

- FOR ARCHITECTURES THAT 'JUMP' TO A CERTAIN HARDWARE ADDRESS UPON RECEIPT OF AN INTERRUPT
- A TASK ENTRY IS ASSOCIATED WITH THE ADDRESS
- PRIORITY IS HIGHER THAN ANY USER-DEFINED

```
task INTERRUPT_HANDLER is
   entry DONE;
   for DONE use at 16#40#;
end INTERRUPT_HANDLER;


task body INTERRUPT_HANDLER is
  begin
    accept DONE do
       ...
    end DONE;
  end INTERRUPT_HANDLER;
```

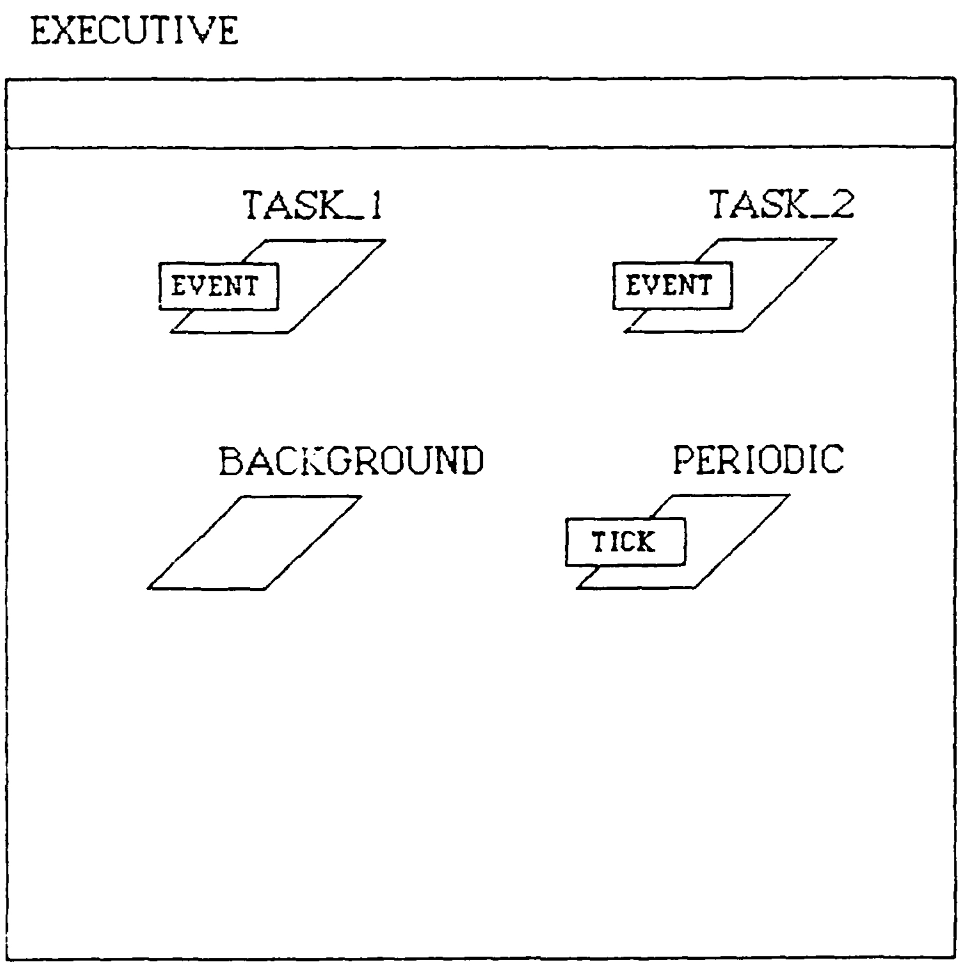
EVENT DRIVEN SYSTEMS W/BACKGROUND

- A cyclic executive might deal with several levels of processing

  -- Event driven processing (high priority, perhaps interrupt handling)

  -- Periodic (cyclic) processing

  -- Background processing (low priority)



```
procedure EXECUTIVE is

    task TASK_1 is
        pragma PRIORITY (10);
        entry EVENT;
    end TASK_1;

    task TASK_2 is
        entry EVENT;
        for EVENT use at 16#110#;
    end TASK_2;

    task BACKGROUND is
        pragma PRIORITY (0);
    end BACKGROUND;

    task PERIODIC is
        pragma PRIORITY (5);
        entry TICK;              -- one tick per cycle
    end PERIODIC;

    task body PERIODIC is
        ...
    begin
        loop
            accept TICK;
            ... -- process a frame
        end loop;
    end PERIODIC;

    -- bodies (or stubs) of other tasks go here

end EXECUTIVE;
```

Second Annual ASEET Symposium

Tutorial on Ada® Exceptions

by
Major Patricia K. Lawlis

lawlis%asu@csnet-relay

Air Force Institute of Technology (AFIT)
and
Arizona State University (ASU)

9 June 1987

® Ada is a registered trademark of the U. S. Government - Ada Joint Program Office

References

- Student Handout, "Ada Applications Programmer - Advanced Ada Software Engineering", USAF Technical Training School, Keesler Air Force Base, July 1986.

- J. G. P. Barnes, Programming in Ada, 2nd edition, Addison-Wesley, 1984.

- Grady Booch, Software Engineering with Ada, Benjamin/Cummings, 1983.

- Theodore F. Elbert, Embedded Programming in Ada, Van Nostrand Reinhold, 1986.

- Putnam P. Texel, Introductory Ada: Packages for Programming, Wadsworth, 1986.

- ANSI/MIL-STD-1815A, "Military Standard - Ada Programming Language" (**LRM**). U. S. Department of Defense, 22 January 1983.

Outline

=> Overview

- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples

Overview

- What is an exception

- Ada exceptions

- Comparison

    - the American way

    - using exceptions

What Is an Exception

- A run time error
- An unusual or unexpected condition
- A condition requiring special attention
- Other than normal processing

Ada Exceptions

- An exception has a name
  - may be predefined
  - may be declared
- The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by **raise** statement
- The exception is handled
  - exception handler may be placed in any **frame**
  - exception propagates until handler is found
  - if no handler anywhere, process aborts

The American Way

```
package Stack_Package is

        type Stack_Type is limited private;

        procedure Push (Stack : in out Stack_Type;
                        Element : in Element_Type;
                        Overflow_Flag : out boolean);
        ...

end Stack_Package;


with Text_IO;
with Stack_Package; use Stack_Package;
procedure Flag_Waving is
        ...
        Stack : Stack_Type;
        Element : Element_Type;
        Flag : boolean;
begin
        ...
        Push (Stack, Element, Flag);
        if Flag then
                Text_IO.Put ("Stack overflow");
                ...
        end if;
        ...
end Flag_Waving;
```

Using Exceptions

```
package Stack_Package is

        type Stack_Type is limited private;
        Stack_Overflow,
        Stack_Underflow : exception;

        procedure Push (Stack : in out Stack_Type;
                        Element : in Element_Type);
                  -- may raise Stack_Overflow
        ...
end Stack_Package;


with Text_IO;
with Stack_Package; use Stack_Package;
procedure More_Natural is
        ...
        Stack : Stack_Type;
        Element : Element_Type;
begin
        ...
        Push (Stack. Element);
        ...
exception
        when Stack_Overflow =>
                Text_IO.Put ("Stack overflow");
        ...
end More_Natural;
```

Outline

- Overview

=> **Naming an exception**

- Creating an exception handler

- Raising an exception

- Handling exceptions

- Turning off exception checking

- Tasking exceptions

- More examples

Naming an Exception

- Predefined exceptions
- Declaring exceptions
- I/O exceptions

Predefined Exceptions

- In package STANDARD (also see chap 11 of LRM)

- CONSTRAINT_ERROR

  violation of range, index, or discriminant constraint...

- NUMERIC_ERROR

  execution of a predefined numeric operation cannot deliver a correct result

- PROGRAM_ERROR

  attempt to access a program unit which has not yet been elaborated...

- STORAGE_ERROR

  storage allocation is exceeded...

- TASKING_ERROR

  exception arising during intertask communication

Declaring Exceptions

exception_declaration ::= identifier_list : **exception**;

- *Exception may be declared anywhere an object declaration* is appropriate

- However, exception is not an object

  - may not be used as subprogram parameter, record or array component

  - has same scope as an object, but its effect may extend beyond its scope

Example:

```
procedure  Calculation  is

        Singular : exception;
        Overflow, Underflow : exception;

begin
    ...
end  Calculation;
```

I/O Exceptions

- Exceptions relating to file processing

- In predefined library unit IO_EXCEPTIONS
  (also see chap 14 of LRM)

- TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO **with** it

```
package IO_EXCEPTIONS is

        NAME_ERROR    : exception;
        USE_ERROR     : exception;     --attempt to use
                                       --invalid operation
        STATUS_ERROR  : exception;
        MODE_ERROR    : exception;
        DEVICE_ERROR  : exception;
        END_ERROR     : exception;     --attempt to read
                                       --beyond end of file
        DATA_ERROR    : exception;     --attempt to input
                                       --wrong type
        LAYOUT_ERROR  : exception;     --for text processing

end IO_EXCEPTIONS;
```

Outline

- Overview
- Naming an exception

=> **Creating an exception handler**

- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples

Creating an Exception Handler

- Defining an exception handler
- Restrictions
- Handler example

Defining an Exception Handler

- Exception condition is "caught" and "handled" by an exception handler

- Exception handler may appear at the end of any frame (block, subprogram, package or task body)

```
begin
    ...
exception
    -- exception handler(s)
end;
```

- Form similar to case statement

exception_handler ::=
    when exception_choice {| exception_choice} **=>**
        sequence_of_statements

exception_choice ::= exception_name | **others**

Restrictions

- Exception handlers must be at the end of a frame

- Nothing but exception handlers may lie between **exception** and **end** of frame

- A handler may name any visible exception declared or predefined

- A handler includes a sequence of statements

    - response to exception condition

- A handler for **others** may be used

    - must be the last handler in the frame

    - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)

    - can be the only handler in the frame

Handler Example

```
procedure Whatever is

        Problem_Condition : exception;

begin
        ...

exception

        when Problem_Condition =>
                Fix_It;

        when CONSTRAINT_ERROR =>
                Report_It;

        when others =>
                Punt;

end Whatever;
```

Outline

- Overview

- Naming an exception

- Creating an exception handler

=> **Raising an exception**

- Handling exceptions

- Turning off exception checking

- Tasking exceptions

- More examples

Raising an Exception

- How exceptions are raised
- Effects of raising an exception
- Raising example

How Exceptions are Raised

- Implicitly by run time system

  - predefined exceptions

- Explicitly by **raise** statement

raise_statement ::= **raise** [exception_name];

  - the name of the exception must be visible at the point of the raise statement

  - a raise statement without an exception name is allowed only within an exception handler

Effects of Raising an Exception

- Control transfers to exception handler at end of frame
  (if one exists)

- Exception is lowered

- Sequence of statements in exception hander is executed

- Control passes to end of frame

- If frame does not contain an appropriate exception handler,
  the exception is propagated

Raising Example

```
procedure Whatever is

        Problem_Condition  : exception;
        Real_Bad_Condition : exception;

begin
        ...
        if Problem_Arises then
                raise Problem_Condition;
        end if;
        ...
        if Serious_Problem then
                raise Real_Bad_Condition;
        end if;
        ...
exception

        when Problem_Condition =>
                Fix_It;

        when CONSTRAINT_ERROR =>
                Report_It;

        when others =>
                Punt;

end Whatever;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception

=> **Handling exceptions**

- Turning off exception checking
- Tasking exceptions
- More examples

Handling Exceptions

- How exception handling can be useful
- Which exception handler is used
- Sequence of statements in exception handler
- Propagation
- Propagation example

How Exception Handling Can Be Useful

- Normal processing could continue if
  - cause of exception condition can be "repaired"
  - alternative approach can be used
  - operation can be retried
- Degraded processing could be better than termination
  - for example. safety-critical systems
- If termination is necessary, "clean-up" can be done first

Which Exception Handler Is Used

- If exception is raised during normal execution, system looks for an exception handler at the end of the frame in which the exception occurred

- If exception is raised during elaboration of the declarative part of a frame

  - elaboration is abandoned and control goes to the end of the frame with the exception still raised

  - exception part of the frame is not searched for an appropriate handler

  - effectively, the calling unit will be searched for an appropriate handler

  - if elaboration of library unit, program execution is abandoned
    - -- all library units are elaborated with the main program

- If exception is raised in exception handler

  - handler may contain block(s) with handler(s)

  - if not handled locally within handler, control goes to end of frame with exception raised

Sequence of Statements in Exception Handler

- Handler completes the execution of the frame
  - handler for a **function** should usually contain a **return** statement
- Statements can be of arbitrary complexity
  - can use most any language construct that makes sense in that context
  - cannot use **goto** statement to transfer into a handler
  - if handler is in a block inside a loop, could use **exit** statement
- Handler at end of package body applies only to package initialization

Propagation

- Occurs if no handler exists in frame where exception is raised

- Also occurs if **raise** statement is used in handler

- Exception is propagated dynamically

  - propagates from subprogram to unit calling it (not necessarily unit containing its declaration)

  - this can result in propagation outside its scope

- Propagation continues until

  - an appropriate handler is found

  - exception propagates to main program (still with no handler) and program execution is abandoned

Propagation Example

```
procedure Do_Nothing is
    --------------
    procedure Has_It is
        Some_Problem : exception;
    begin
        ...
        raise Some_Problem;
        ...
    exception
        when Some_Problem =>
            Clean_Up;
            raise;
    end Has_It;
    --------------
    procedure Calls_It is
    begin
        ...
        Has_It;
        ...
    end Calls_It;
    --------------
begin  -- Do_Nothing
    ...
    Calls_It;
    ...
exception
    when others => Fix_Everything;
end Do_Nothing;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions

=> Turning off exception checking

- Tasking exceptions
- More examples

Turning Off Exception Checking

- Overhead vs efficiency
- Pragma SUPPRESS
- Check identifiers

Overhead vs Efficiency

- Exception checking imposes run time overhead

  - interactive applications will never notice
  - real-time applications have legitimate concerns but must not sacrifice system safety

- When efficiency counts

  - first and foremost, make program work
  - be sure possible problems are covered by exception handlers
  - check if efficient enough - stop if it is
  - if not, study execution profile
    - -- eliminate bottlenecks
    - -- improve algorithm
    - -- avoid "cute" tricks
  - check if efficient enough - stop if it is
  - if not, trade-offs may be necessary
  - some exception checks may be expendable since debugging is done
  - however, every suppressed check poses new possibilities for problems
    - -- must re-examine possible problems
    - -- must re-examine exception handlers
  - always keep in mind
    - -- problems <u>will</u> happen
    - -- critical applications must be able to deal with these problems

Moral

Improving the algorithm is far better - and easier in the long run - than suppressing checks

Pragma SUPPRESS

- *Only allowed immediately within a declarative part or* immediately within a package specification

pragma SUPPRESS (identifier [,[**ON** =>] name]);

- identifier is that of the check to be omitted (next slide lists identifiers)

- *name* is that of an object, type, or unit for which the check is to be suppressed

  -- if no name is given, it applies to the remaining declarative region

- An implementation is free to ignore the suppress directive for any check which may be impossible or too costly to suppress

Example:

```
pragma SUPPRESS (INDEX_CHECK, ON => Index);
```

Check Identifiers

- These identifiers are explained in more detail in chap 11 of the LRM

- Check identifiers for suppression of CONSTRAINT_ERROR checks

  ACCESS_CHECK
  DISCRIMINANT_CHECK
  INDEX_CHECK
  LENGTH_CHECK
  RANGE_CHECK

- Check identifiers for suppression of NUMERIC_ERROR checks

  DIVISION_CHECK
  OVERFLOW_CHECK

- Check identifier for suppression of PROGRAM_ERROR checks

  ELABORATION_CHECK

- Check identifier for suppression of STORAGE_ERROR check

  STORAGE_CHECK

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking

=> **Tasking exceptions**

- More examples

Tasking Exceptions

- Exception handling is trickier for tasks
- Exceptions during task rendezvous
- Tasking example

Exception Handling Is Trickier for Tasks

- Rules are not really different, just more involved
    - local exceptions handled the same within frames

If exception is raised

- during elaboration of task declarations
    - the exception TASKING_ERROR will be raised at the *point of task activation*
    - the task will be marked completed
- during execution of task body (and not resolved there)
    - task is completed
    - exception is <u>not</u> propagated
- during task rendezvous
    - this is the really tricky part

Exceptions During Task Rendezvous

- If the **called** task terminates abnormally

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

- If the **calling** task terminates abnormally

  no exception propagates to the **called** task

- If an exception is raised in **called** task within an **accept** (and not handled there locally)

  the same exception is raised in the **calling** task at the point of the entry call
  (even if exception is later handled outside of the accept in the called task)

- If an entry call is made for entry of a task that becomes completed before accepting the entry

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

Tasking Example

```
procedure  Critical_Code  is

        Failure : exception;
         -------------
        task  Monitor  is
                entry Do_Something;
        end Monitor;
        task  body  Monitor  is
        ...
        begin
                accept  Do_Something   do
                        ...
                        raise  Failure;
                        ...
                end   Do_Something;
                ...
        exception   -- exception handled here
                when Failure =>
                        Termination_Message;
        end Monitor;
        -------------
begin  -- Critical_Code
        ...
        Monitor.Do_Something;
        ...
exception   -- same exception will be handled here
        when Failure =>
                Critical_Problem_Message;

end   Critical_Code;
```

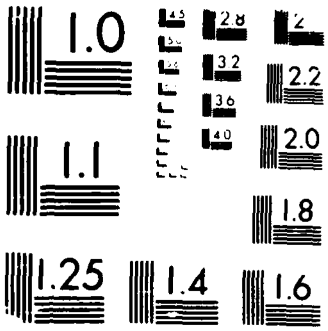
Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions

=> **More examples**

AD-A190 366 TUTORIAL TRACK II ADVANCED ADA TOPICS(U) INFORMATION 2/2
SYSTEMS AND TECHNOLOGY CENTER W-P AFB OH ADA VALIDATION
FACILITY P LAWLIS ET AL 09 JUN 87

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

[illegible] TANDARDS 1963 A

Interactive Data Input

```
with Text_io; use Text_io;
procedure Get_Input (Number : out integer) is

        type Input_Type is integer range  0..100;
        package Int_io is new Integer_io (Input_Type);
        In_Number : Input_Type;

begin  -- Get_Input

        loop              -- to try again after incorrect input

                begin  -- inner block to hold exception handler

                        put ("Enter a number 0 to 100");
                        Int_io.get (In_Number);
                        Number := In_Number;
                        exit;  -- to exit loop after correct input

                exception
                        when DATA_ERROR | CONSTRAINT_ERROR =>
                                put ("Try again, fat fingers!");
                                Skip_Line;    -- must clear buffer

                end;  -- inner block

        end  loop;

end  Get_Input;
```

Propagating Exception Out of Scope and Back In

```
declare
        package Container is
                procedure Has_Handler;
                procedure Raises_Exception;
        end Container;
          ----------
        procedure Not_in_Package is
        begin
                Container.Raises_Exception;
        exception
                when others => raise;
        end  Not_in_Package;
          ----------
        package  body  Container  is
                Crazy : exception;
                procedure  Has_Handler  is
                begin
                        Not_in_Package;
                exception
                        when Crazy => Tell_Everyone;
                end  Has_Handler;
                procedure   Raises_Exception  is
                begin
                        raise Crazy;
                end   Raises_Exception;
        end Container;
begin
        Container.Has_Handler;
end;
```

Keeping a Task Alive

```
task Monitor is
    entry Do_Something;
end Monitor;

task body Monitor is
begin
    loop        -- for never-ending repetition
        ...
        select
            accept Do_Something do

                begin  -- block for exception handler
                    ...
                    raise Failure;
                    ...
                exception
                    when Failure => Recover;
                end;  -- block

            end Do_Something;  -- exception must be
                               -- lowered before exiting
            ...
        end select;
        ...
    end  loop;

exception
    when others =>
        Termination_Message;
end Monitor;
```

END
DATE
FILMED
5-88
DTIC